# Rapport TEMPS REEL

UF Système concurrents et temps réel

**BAH Amadou Thierno** (conception, code watchdog, code caméra, code batterie, code de gestion de threads ) – 4IR SC

**BENCHEHIDA Yacine** (conception, code création threads, mutex et sémaphores, compte-rendu, vidéos ) – 4IR SC

**AL AJROUDI Alexandre** (conception, code de réinitialisation du robot, code caméra, compte-rendu) – 4IR SC

*Lien git* : https://github.com/tabha/real-time.git

# Table des matières

# Introduction

Dans un premier temps, nous présenterons la conception en trois axes, l'une focalisée sur la communication entre le moniteur et le superviseur, la seconde consacrée au contrôle du robot et la troisième au traitement vidéo.

Dans un deuxième temps, nous verrons la transformation *AADL* vers *Xenomai*, notamment la partie thread du projet, la partie partage de données, le port d'événement ainsi que le port d'événement de données.

Enfin, nous présenterons l'analyse ainsi que la validation de toutes les fonctionnalités du cahier des charges.

# I. Conception

## 1) Diagramme fonctionnel général

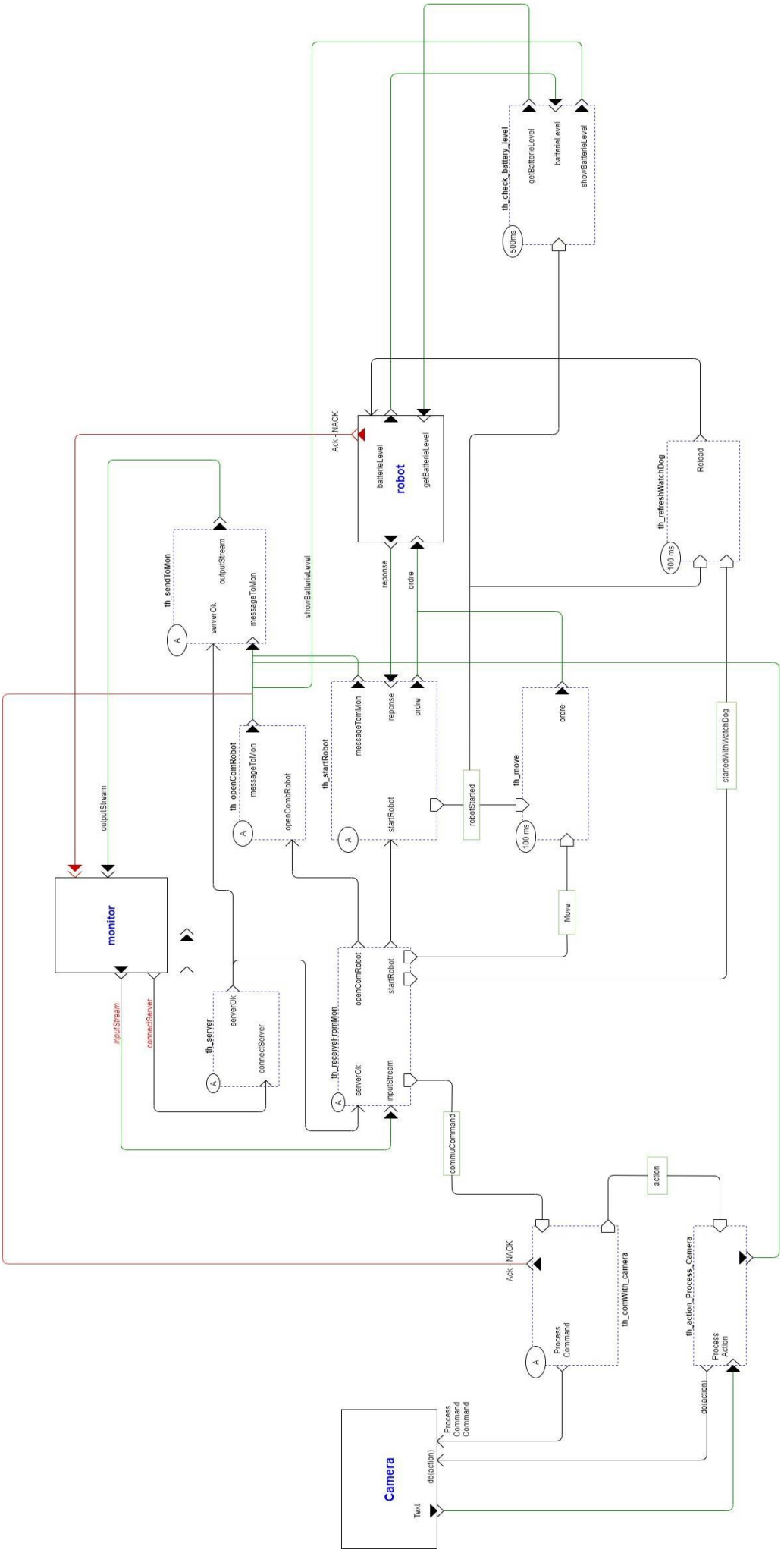



**Figure 1 :** *Diagramme fonctionnel du système*

## 2) Groupe de threads gestion du moniteur

### a) Diagramme fonctionnel du groupe gestion du moniteur

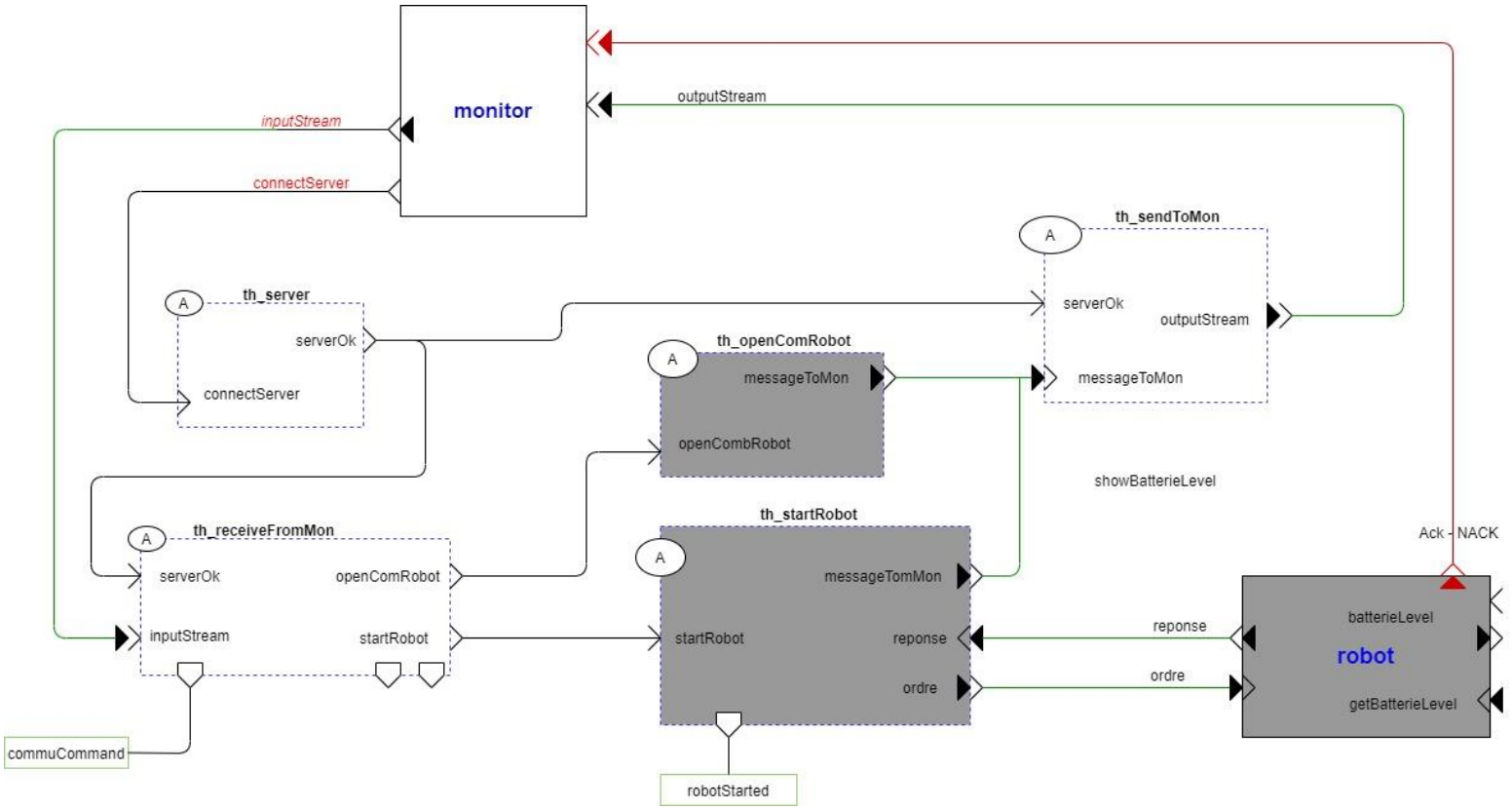



**Figure 2 :** *Diagramme fonctionnel du groupe gestion du moniteur*

Ce diagramme fonctionnel illustre bien les trois threads essentiels au bon fonctionnement du groupe gestion du moniteur.

- Le thread « **th_server** » permet au superviseur de se connecter.
- Le thread « **th_receiveFromMon** » permet de recevoir un message en provenance du moniteur.
- Enfin, le thread « **th_sendToMon** » permet, au contraire, d'envoyer un message au moniteur.

### b) Diagrammes d'activité du groupe gestion du moniteur

Le thread ***receiveFromMon*** permet de spécifier les actions à effectuer en fonction du message reçu :

- Si ***MESSAGE_MONITOR_LOST*** est reçu alors cela signifie que la connexion avec le moniteur est perdue, on ferme la connexion active avec le robot ainsi que le moniteur avant de redémarrer le serveur moniteur.

- Si ***MESSAGE_ROBOT_COM_OPEN*** est reçu alors cela signifie qu'on reçoit une demande d'ouverture de communication. On rend disponible le sémaphore *sem_openComRobot*.
- Si ***MESSAGE_ROBOT_GO(DIRECTION)*** est reçu alors cela signifie qu'on reçoit un ordre de mouvement et que par conséquent on modifie la variable move pour prendre la direction.
- Si ***MESSAGE_ROBOT_START*** est reçu alors on accède au sémaphore sem_startRobot et on initialise la variable watchDog à l'état STATE

- Si ***MESSAGE_CAMERA_DO_ACTION*** est reçu alors cela signifie qu'on reçoit un ordre de direction et on instancie la variable commandCamear en fonction de la direction optée.

Enfin, après avoir reçu les différents messages, on libère le buffer avec le message FREE MEMORY

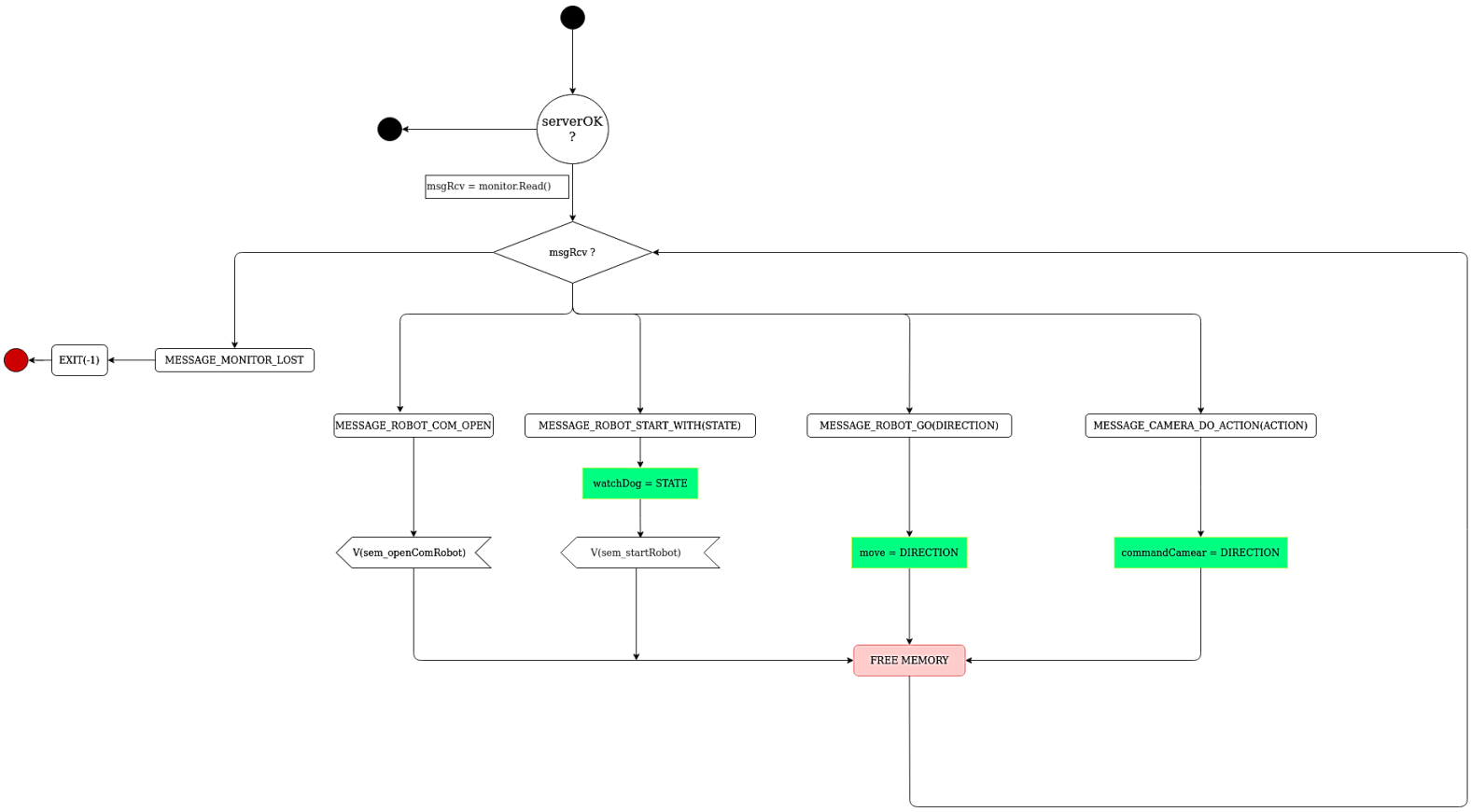



**Figure 3 :** *Diagramme d'activité du thread* ***th_receiveFromMon***

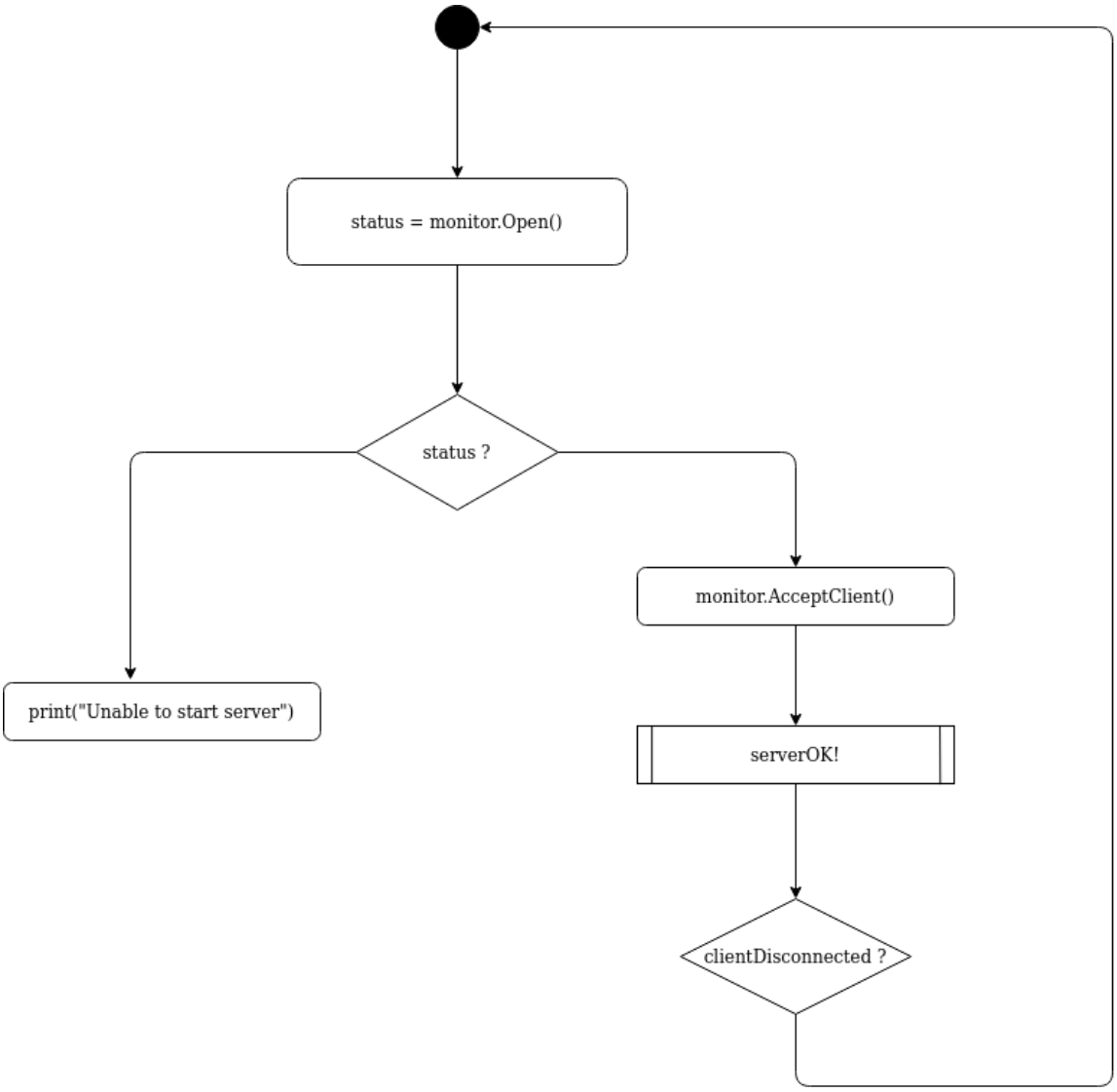



**Figure 4 :** *Diagramme d'activité du thread **th_server***

Le thread ***th_server*** permet de lancer un serveur pour établir la communication avec le moniteur. La méthode *Open* de la classe *ComMonitor* va demander l'ouverture du serveur. Si *status* est inférieur à 0 : impossible de démarrer le serveur sur le port sinon le serveur passe en état d'attente du client moniteur.

Le thread ***th_sendToMon*** va permettre d'envoyer un message au moniteur. Dans un premier temps, le thread est en attente du sémaphore sem_serverOk qui lui signale la parfaite connexion entre le serveur et le moniteur.

Si ce n'est pas le cas, le traitement consiste à placer le message dans la file et l'envoyer via l'appel de fonction monitor.Write() en supposant que le socket est disponible via mutex_monitor.

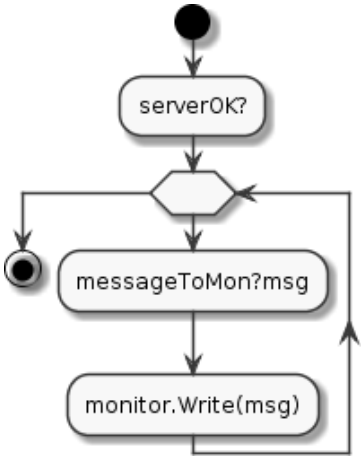
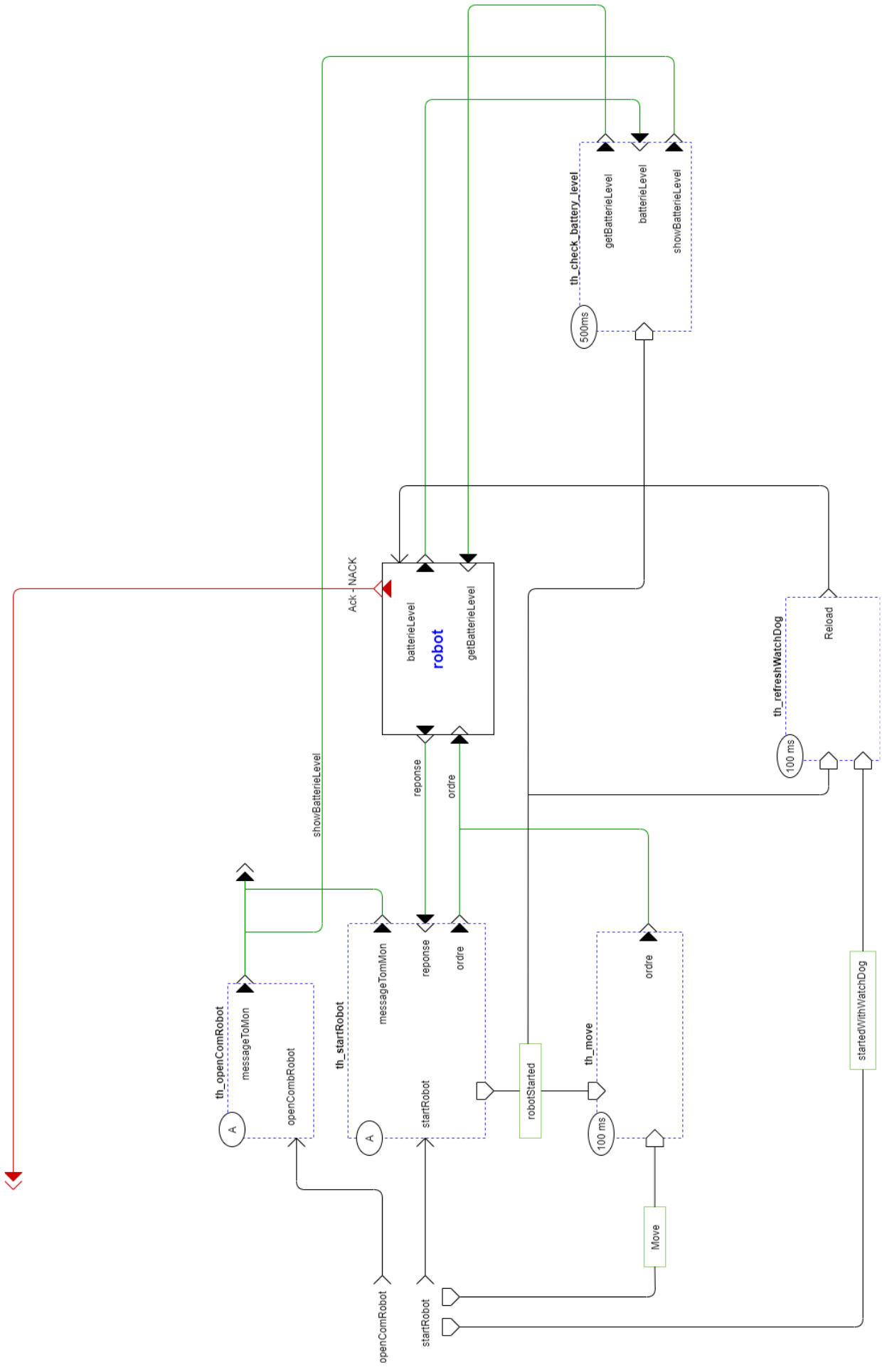



**Figure 5 :** *Diagramme d'activité du thread **th_sendToMon*** →

3) Groupe de threads gestion du robot

a) Diagramme fonctionnel du groupe gestion robot



Ce diagramme fonctionnel énumère tous les threads permettant le bon fonctionnement du groupe de gestion du robot :

- Le thread ***th_check_batery_level*** a pour but d'envoyer au moniteur tous les 500 ms l'état de la batterie du robot

- Le thread ***th_move*** permet de commander les mouvements du robot de façon périodique

- Le thread ***th_startRobot*** a pour mission d'activer le démarrage du robot. En effet, il peut avoir 2 actions en fonction de ce qu'il reçoit. Soit un démarrage sans *watchdog* qui a pour finalité uniquement d'allumer le robot. Soit un démarrage avec *watchdog*, qui en plus d'allumer le robot, lance simultanément le thread ***th_refreshWatchdog***

- Le thread ***th_openComRobot*** qui a pour fonction d'ouvrir une communication avec le robot

- Le thread ***th_refreshWatchdog,*** qui après avoir été lancé par le thread ***th_startRobot***, met en place le *watchdog* tous les 100ms

b) Diagrammes d'activité du groupe de gestion robot

Le thread ***th_check_batery_level*** permet donc d'envoyer périodiquement au moniteur l'état de la batterie. Si la variable *robotStarted*, disponible grâce au mutex « rt_mutex_acquire(..) », est égale à 1 alors le message est mis dans la file d'attente pour être envoyé au moniteur.

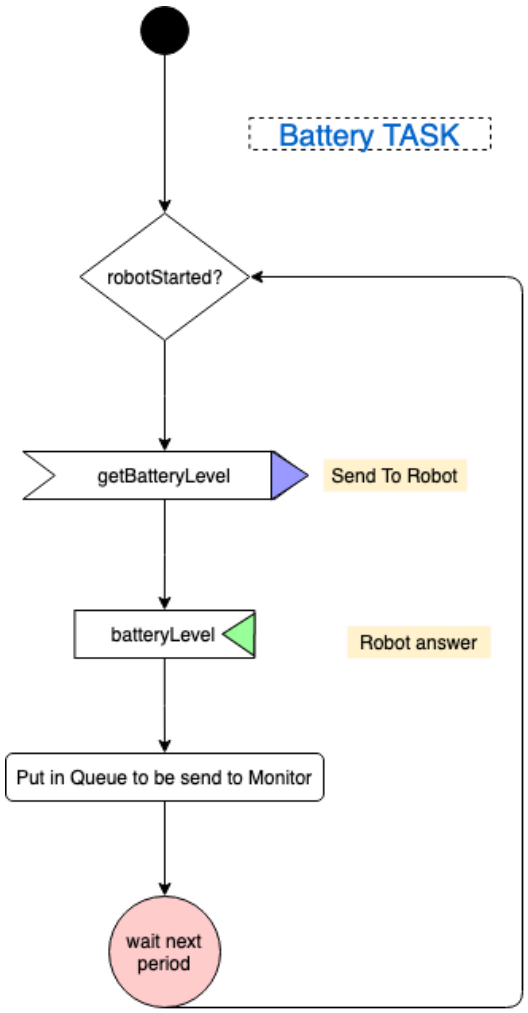
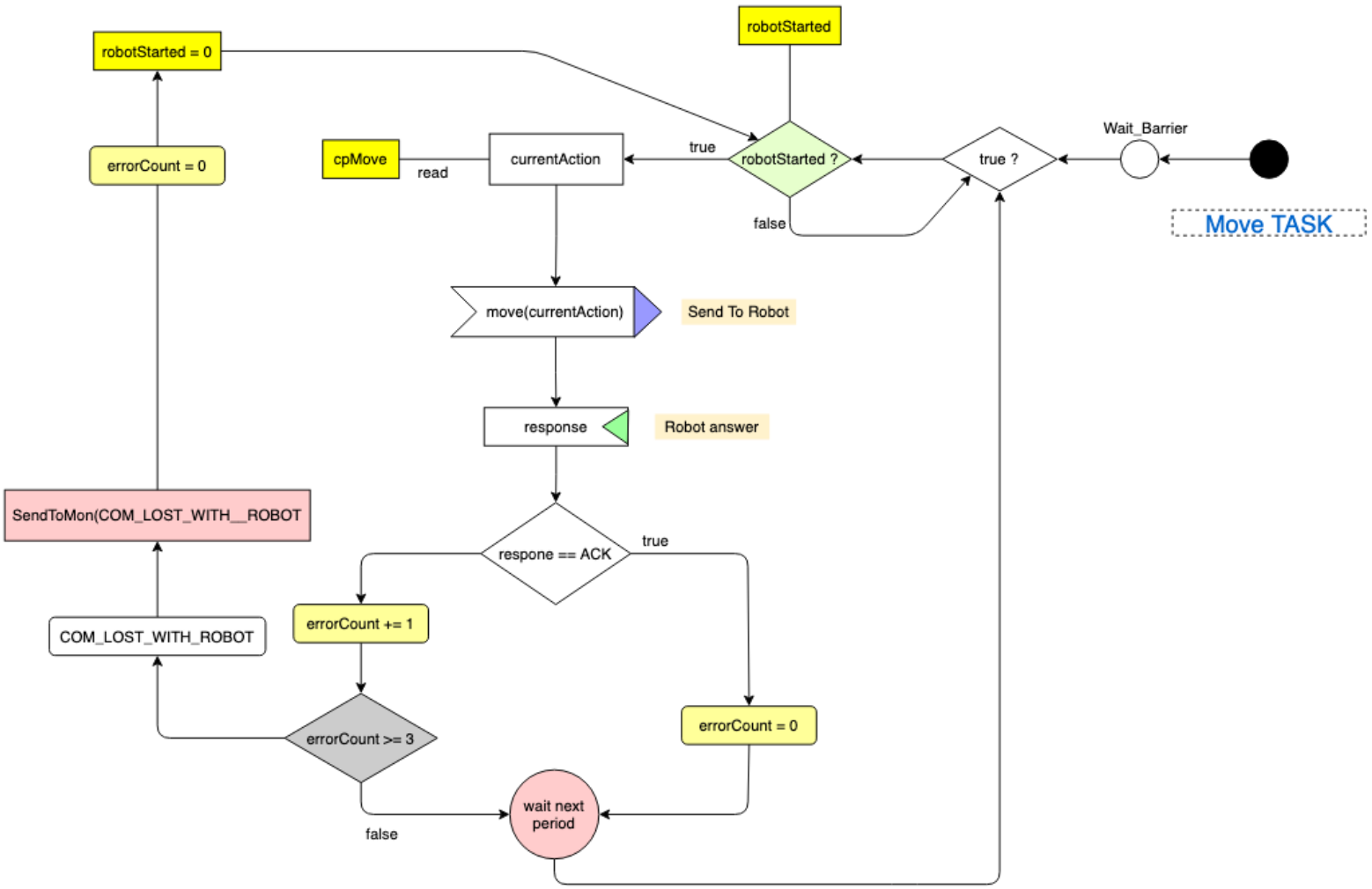



**Figure 5:** *Diagramme d'activité du thread* ***th_check_batery_level***




Le thread ***th_move*** a pour mission d'envoyer via le superviseur les messages de déplacements (reçu du moniteur) directement au robot en supposant celui-ci allumé.

**Figure 6 :** *Diagramme d'activité du thread **th_move***

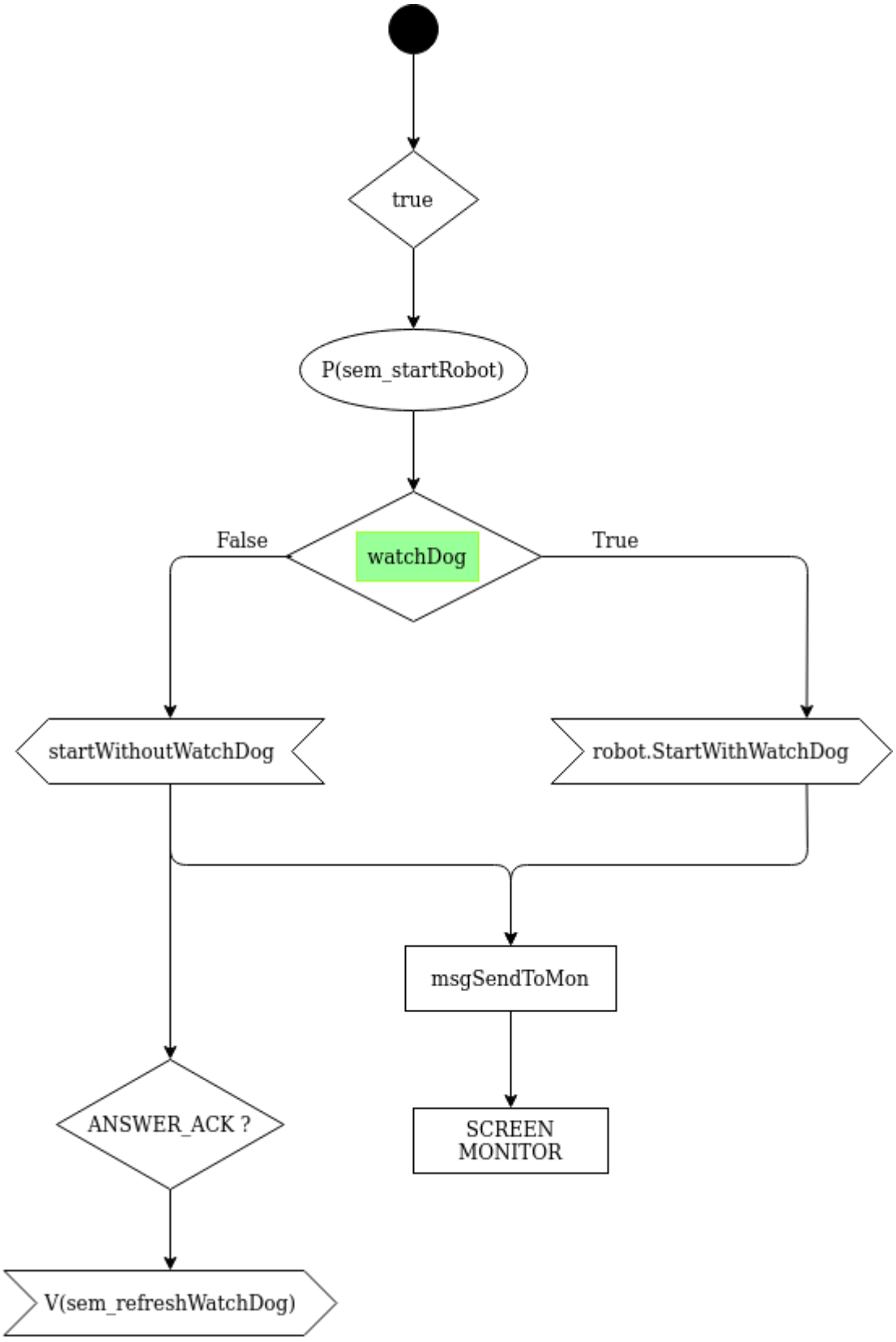



**Figure 7 :** *Diagramme d'activité du thread* ***th_startRobot***

Le thread ***th_startRobot*** a pour but de démarrer le robot. Un démarrage avec ou sans *watchdog* est géré par le lancement ou non du thread ***th_refreshWatchdog***.

Le thread ***th_openComRobot*** permet l'ouverture d'une communication avec le robot.

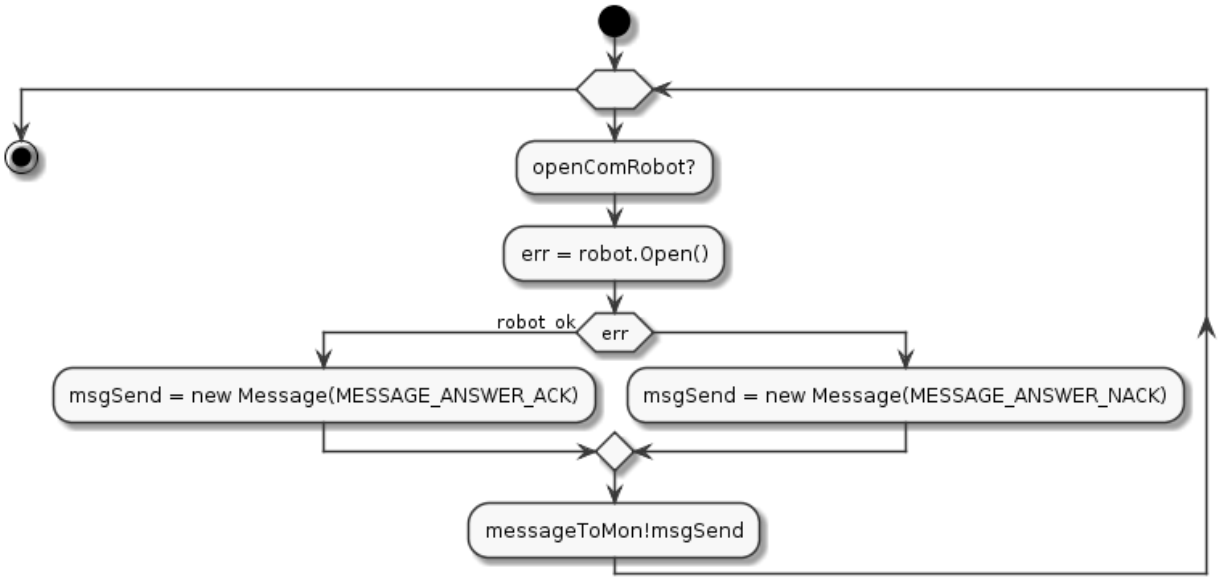



**Figure 8 :** *Diagramme d'activité du thread* ***th_openComRobot***

Le thread ***th_refreshWatchdog*** permet, après avoir été débloqué par le thread ***th_startRobot***, de s'assurer que le robot soit actif en réinitialisant périodiquement son timer *watchdog*.

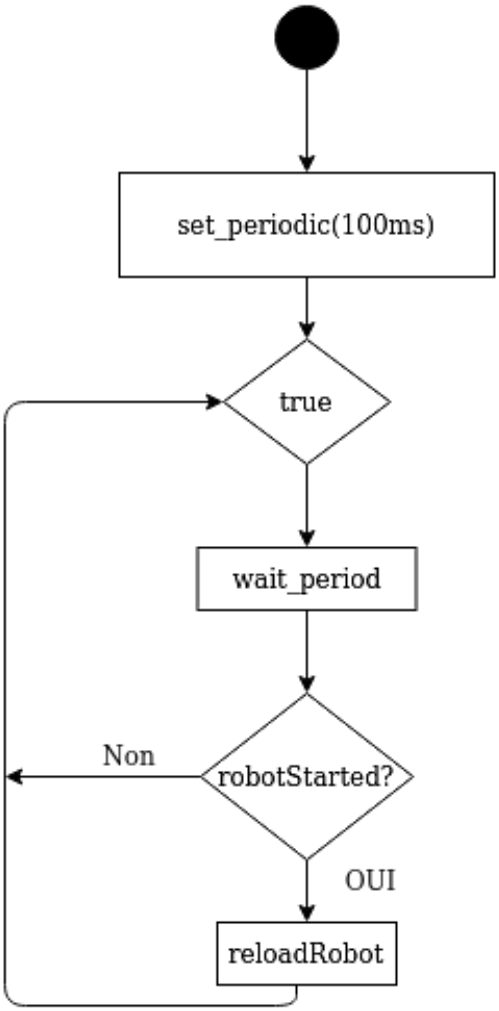


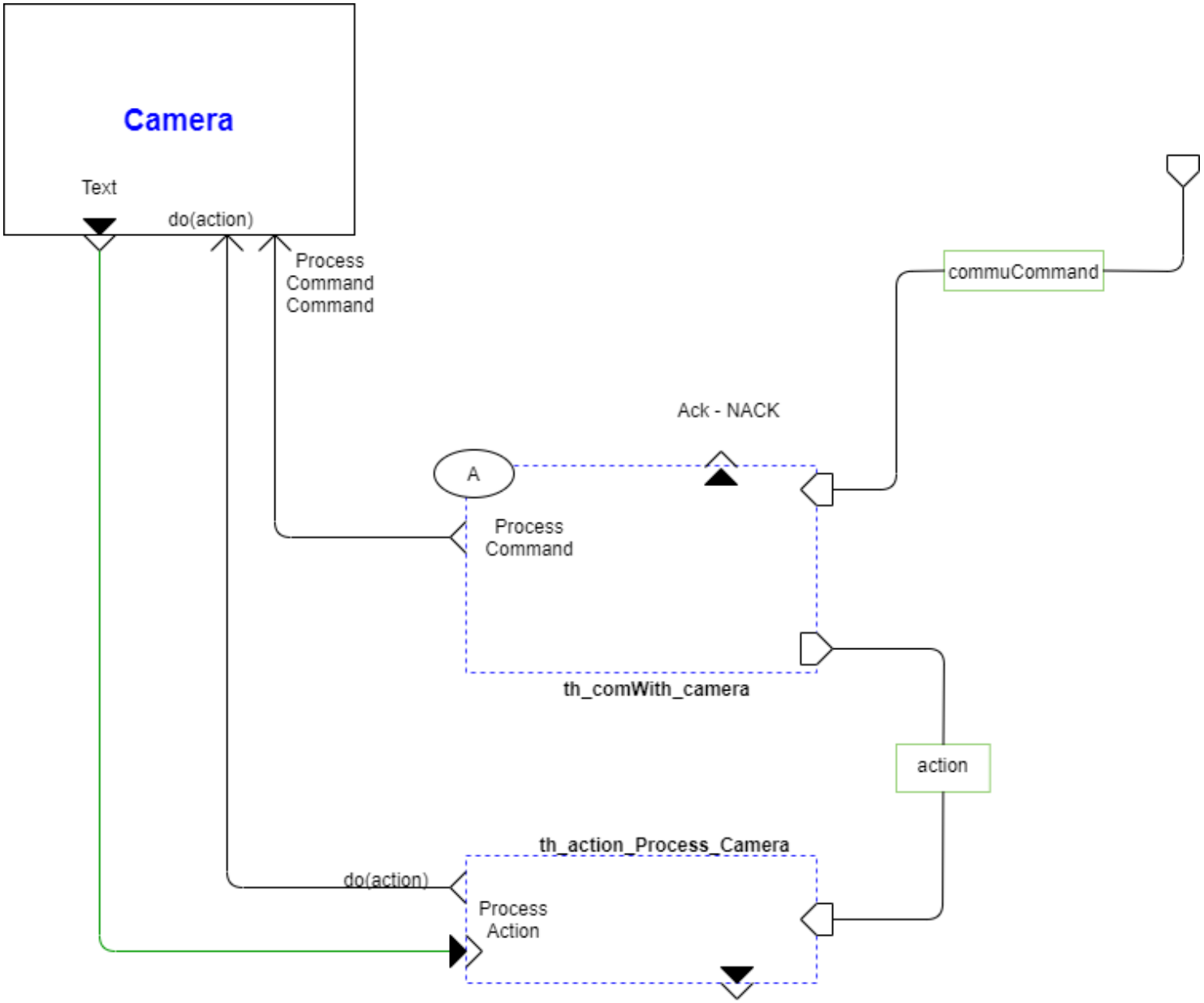

**Figure 9 :** *Diagramme d'activité du thread* ***th_refreshWatchdog***

## 4) Groupe de threads vision

### a) Diagramme fonctionnel du groupe vision

Ce diagramme fonctionnel énumère tous les threads permettant le bon fonctionnement du groupe de gestion vision :

- Le thread ***th_comWith_camera*** permet la communication avec la caméra.
- Le thread ***th_action_Process_Camera*** va récupérer, traiter, envoyer les images de la caméra et s'occuper de l'extinction du robot.



b) Diagrammes d'activité du groupe de gestion vision

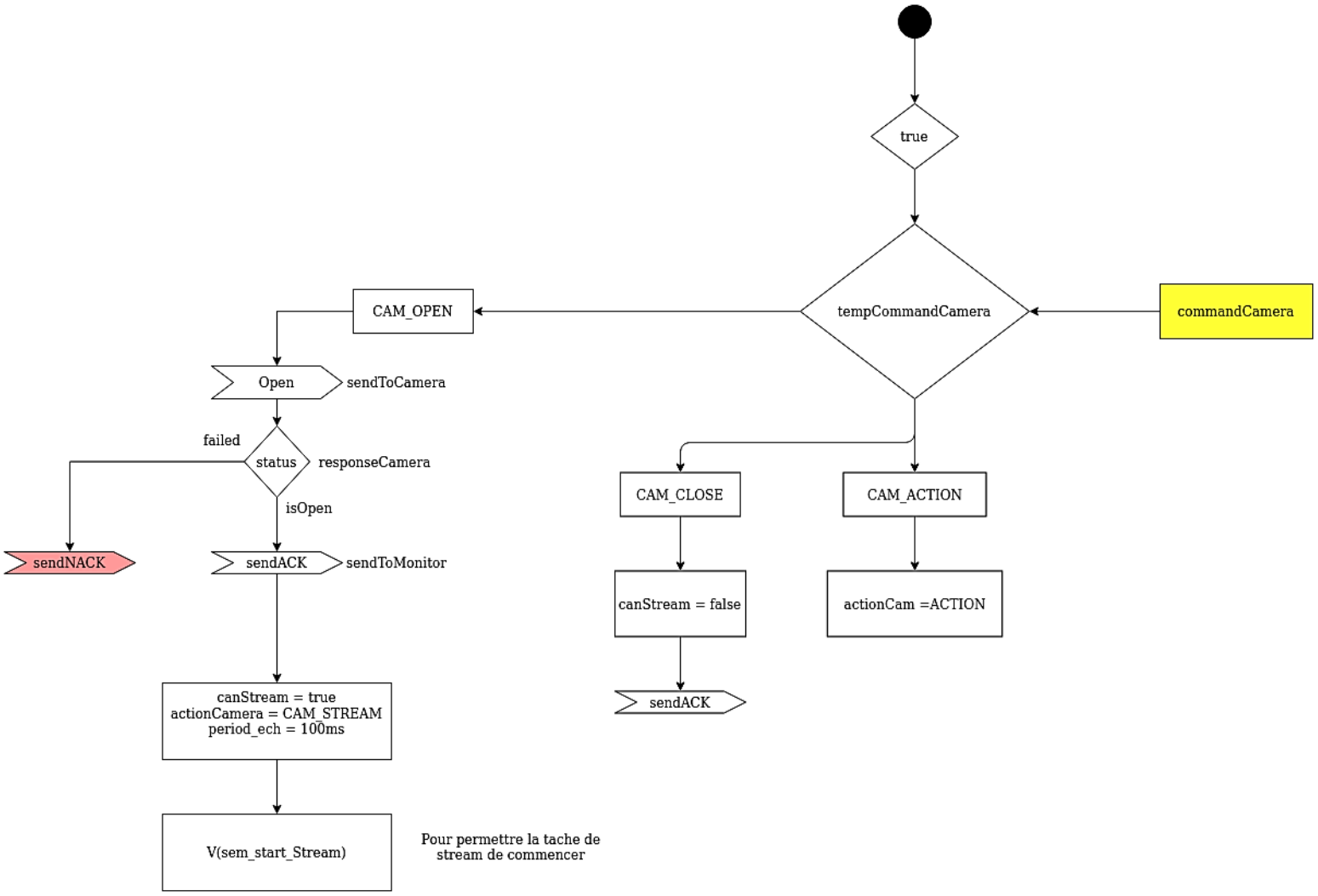



**Figure 10 :** *Diagramme d'activité du thread **th_comWith_camera***

# II. Transformation AADL vers Xenomai

## 1) Thread

### a) Instanciation et démarrage

Chaque thread a été implémenté par un RT_TASK lui-même déclaré dans le fichier tasks.h. La méthode Init() de la classe *Tasks* permet de créer les différents threads au démarrage du programme :

```
if (err = rt_task_create(&th_server, "th_server", 0, PRIORITY_TSERVER, 0))
{
        cerr << "Error task create: " << strerror(-err) << endl << flush;
        exit(EXIT_FAILURE);
    }
```

La fonction *rt_Task_create* initialise le thread *th_server* en lui attribuant un nom permettant de l'identifier lors des fonctions d'affichage à la console, mais aussi spécifie la méthode utilisée lors de son exécution et lui affecte une priorité.

La méthode Run() va quant à elle démarrer les threads initialisés plus tôt dans la méthode init(). La méthode rt_task_start permet d'associer chaque thread à une méthode.

Par exemple, au niveau du thread th_server, la méthode associée est Server_Task (ligne 220) :

```
if (err = rt_task_start(&th_server, (void(*)(void*)) & Tasks::ServerTask, t
his)) {
        cerr << "Error task start: " << strerror(-err) << endl << flush;
        exit(EXIT_FAILURE);
    }
```

Par analogie, tous les threads sont associés à une méthode de la même façon.

### b) Code à exécuter

*Comment se fait le lien sous Xenomai entre le thread et le traitement à exécuter ?*

Le lien sous Xenomai entre le thread et le traitement à exécuter est mis en place à travers la méthode auquel un thread va s'appliquer et est spécifié dans l'appel à rt_task_start. Chaque méthode associée à un thread, implémente une boucle infinie pendant laquelle le traitement de la tâche s'effectue.

Par exemple, au niveau du thread th_server (ci-dessus), on observe clairement ce qui permet de faire le lien sous Xenomai entre le thread et le traitement à exécuter : ***rt_task_start(&th_server, (void(*)(void*)) & Tasks::ServerTask, this))***

*Remarque :* Le traitement à exécuter est implémenté dans la procédure : *void Tasks::ServerTask(void *arg)* de la ligne 288 à 310.

### c) Niveau de priorités

Expliquer comment vous fixez sous Xenomai le niveau de priorité d'un thread AADL.

Sous Xenomai, les threads détenant la priorité la plus haute sont conduits, par l'ordonnanceur, dans les files prioritaires prêtes à s'exécuter. La priorité d'un thread est définie en fonction du rôle joué par celui-ci avec les autres threads et son interdépendance avec eux (exemple ***th_refreshWatchdog*** dépend du ***th_server***).

Le niveau de priorité est fixé par l'attribution d'un entier entre **0 et 99** avec **99** signifiant la priorité la plus importante.

d) Action périodique

Expliquez comment vous rendez périodique l'activation d'un thread AADL sous Xenomai

Pour rendre un thread périodique, on utilise les méthodes *rt_task_set_periodic* et *rt_task_wait_period*.

La méthode rt_task_set_periodic définit **la période en ticks d'horloge**, en utilisant la fonction de conversion nanosecondes vers ticks d'horloge de Xenomai, ainsi que **le début de la périodicité**. Le paramètre TM_NOW spécifie le comportement périodique à la suite de l'appel de fonction.

Lors du traitement périodique, la méthode *rt_task_wait_period* permet de faire attendre le thread jusqu'au prochain cycle, c'est-à-dire le temps d'une période. Cela va s'exprimer en implémentant une boucle *while* qui va exécuter le code après avoir patienté le temps d'une période :

```
//Code périodique avec une période de 100 ms
rt_task_set_periodic(&th_refreshWatchDog, TM_NOW, rt_timer_ns2ticks(1000000
00));

    while(1) {
        rt_task_wait_period(NULL);
        /*Code périodique de la tâche refreshWatchDog
}
```

## 2) Données partagées

a) Instanciation

*Quelle structure instancie une donnée partagée ?*

Chaque donnée partagée a été instanciée comme une variable globale dans tasks.h. Un mutex est associé à chaque donnée partagée. Son rôle est de protéger ces données visant à garantir et préserver la validité et l'exactitude des valeurs stockées dans chaque variable globale.

Ci-dessous un exemple de déclaration d'une variable ainsi du mutex associé :

```
ComRobot robot;
RT_MUTEX mutex_robot;
```

b) Accès en lecture et écriture

*Comment garantissez-vous sous Xenomai l'accès à une donnée partagée ?*

En effet, la lecture ou l'écriture de la donnée à un instant t, par une unique entité garantit l'intégrité de cette variable partagée.

L'accès à une donnée partagée est sous la protection d'un mutex. Pour accéder à une donnée partagée en lecture ou en écriture, il suffit de verrouiller le mutex associé à cette donnée. Ce processus de verrouillage ne peut être possible qu'une seule fois pour un mutex associé. Par hasard, si un deuxième thread lance la fonction *rt_mutex_acquire* alors que le mutex est verrouillé : il est mis dans un état d'attente de libération du mutex.

Le fonctionnement du thread ***th_check_batery_level*** exige d'accéder à la valeur de la *variable partagée robotStarted*. Cela est possible grâce à l'utilisation d'un mutex :

```
//cout << "Periodic movement update";
        rt_mutex_acquire(&mutex_robotStarted, TM_INFINITE);
        rs = robotStarted;
        rt_mutex_release(&mutex_robotStarted);
```

Après exécution de cette portion de code, la *variable partagée robotStarted* est lue par le thread ***th_check_batery_level*** seulement lorsque le mutex est verrouillé. Enfin, le mutex est libéré après lecture de la variable partagée grâce à la fonction *rt_mutex_release().*

## 3) Port d'événement

a) Instanciation

*Comment avez-vous instancié un port d'événement ?*

Un port d'évènement a été instancié par un sémaphore (RT_SEM) déclaré dans le fichier tasks.h. La création d'un sémaphore est effectuée via la fonction *rt_sem_create* dans le fichier tasks.cpp. La méthode init() initialise les structures de l'application dont tous les sémaphores exemple :

```
 if (err = rt_sem_create(&sem_CamCommunication, NULL, 0, S_FIFO)) {
        cerr << "Error semaphore create: " << strerror(-
err) << endl << flush;
        exit(EXIT_FAILURE);
    }
```

b) Envoi d'un événement

*Quels services ont été employés pour signaler un événement ?*

Pour signaler un événement, on emploie le service RT_SEM_V pour envoyer l'événement sous la forme d'un signal destiné à une seule tâche. Pour réaliser l'événement

refreshWatchdog, on informe le thread ***th_refreshWatchdog*** qu'il peut s'exécuter dans le fichier taskp.cpp à la ligne 490 :

```
rt_sem_v(&sem_refreshWatchDog);
```

De plus, on peut aussi employer le service RT_SEM_BROADCAST pour envoyer l'événement à toutes les tâches en attente. Ce service permet de débloquer toutes les tâches en état d'attente et de notifier à ces tâches la réalisation de l'événement.

Prenons par exemple, la fonction RT_SEM_BROADCAST, dans le fichier tasks.cpp, est appelée avec comme paramètre l'adresse du sémaphore *sem_serverOk*. En effet, cette fonction lance les threads *SendToMon* et *ReceiveFromMon* qui sont en attente de l'activation du serveur.

```
rt_sem_broadcast(&sem_serverOk);
```

c) Réception d'un évènement

*Comment se fait l'attente d'un événement ?*

L'attente d'un événement est gérée par la fonction rt_sem_p(). Périodiquement, cette fonction tente de récupérer un jeton du sémaphore dans le but d'exécuter la suite du code.

La fonction rt_sem_p() est appelée avec le paramètre TM_INFINITE qui indique que l'attente du sémaphore peut être infinie et par conséquent ne jamais s'effectuer. De cette façon le thread reste infiniment bloqué tant qu'il n'a pas reçu de notification d'un nouvel événement.

**4) Port d'événement-données**

a) Instanciation

*Donnez la solution retenue pour implémenter un port d'événement-données avec Xenomai*.

Pour implémenter un port d'événement-données, la solution retenue est l'utilisation d'une file de messages. La création de file de messages est gérée par la primitive rt_queue_create dans le fichier tasks.cpp.

Pour l'événement-données q_messageToMon, on a donc les étapes suivantes :

- Déclaration de la variable globale : RT_QUEUE q_messageToMon, dans le fichier tasks.h : *RT_QUEUE q_messageToMon;*

- Création de l'événement-donnée avec la fonction rt_queue_create dans le fichier tasks.cpp

```
rt_queue_create(&q_messageToMon, "q_messageToMon", sizeof (Message*)*50, Q_
UNLIMITED, Q_FIFO))
```

b) Envoi d'une donnée

*Quels services avez-vous employés pour envoyer des données ?*

Pour envoyer des données, on a utilisé une file d'attente (pour l'écriture des données) associée à la fonction WriteInQueue() ci-dessous.

```
WriteInQueue(&q_messageToMon, msgSend);
```

c) Réception d'une donnée

*Quels services avez-vous employés pour recevoir des données ?*

L'envoie de donnée entre les threads a été permise principalement grâce à l'utilisation de deux méthodes :

- Une méthode asynchrone utilisant une variable partagée protégée par un mutex
- Une méthode synchrone utilisant une file associée à la méthode ReadInQueue() avec une lecture bloquante.

```
 if ((err = rt_queue_read(queue, &msg, sizeof ((void*) &msg), TM_INFINITE))
 < 0) {
        cout << "Read in queue failed: " << strerror(-
err) << endl << flush;
        throw std::runtime_error{"Error in read in queue"};
    }
```

# III. Analyse et validation de la conception

Nous avons réalisé une vidéo récapitulative afin de mettre en exergue le correct fonctionnement de l'ensemble des fonctionnalités répondant au cahier des charges.

Vous trouverez cette vidéo dans le git du projet (https://github.com/tabha/real-time.git).

Cette vidéo a été réalisé avec beaucoup d'amour, en espérant embellir votre journée !

# CONCLUSION

A travers ce bureau d'études, nous nous sommes approprié les concepts clés liés à la conception d'une application concurrente en temps réel. Ce qui nous a permis de monter en compétence sur la programmation objet, en particulier sur les langages C et C++ ainsi que la conception d'application en AADL et Xenomai.

Ainsi, nous avons trouvé ce projet très formateur, aussi bien sur le plan technique mais aussi sur le plan humain, et notamment dans la gestion de projet, ce qui nous servira sans aucun doute dans notre avenir professionnel.

Enfin, nous souhaitons remercier Monsieur HLADIK ainsi que Monsieur LUBAT de nous avoir encadré et de nous avoir aidé pour la réalisation de ce projet.